QSORT

Divide — And — Conquer

Algorithm of QSORT

C_Language of QSORT

Running Time of QSORT

DIVIDE – AND – CONQUER

- An algorithm to make a problem be small ,but similar to original problem.

- **Divide :** Partition the array[p…r] into two subarray a[p…q]___and a[q+1…r] .

- **Conquer :** Sort the two subarray A[p…q] and a[q+1…r] by recursive calls to qsort .

- **Combine :** When each subarray are sorted in place ,the original array is sorted completely . As the result, it’s no need to combine them.

ALGORITHM OF QSORT

2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4
2	1	7	8	3	5	6	4
2	1	3	8	7	5	6	4
2	1	3	8	7	5	6	4
2	1	3	8	7	5	6	4
2	1	3	4	7	5	6	8

C_LANGUAGE OF QSORT[1]

```
Qsort (A,p,r)

if p<r
    then q ← Patition (A,p,r)
        Qsort (A,p,q-1)
        Qsort(A,q+1,r)
```

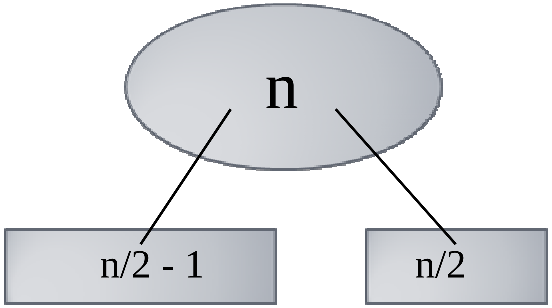
C_LANGUAGE OF QSORT[2]

Partition (A,p,r)

```
x ← A[r]
i ← p-1
for  j←p to r-1
     do if  A[j]<=x
          then i←i+1
               exchange A[i]←→A[j]
exchange A[i+1]←→A[r]
return i+1
```

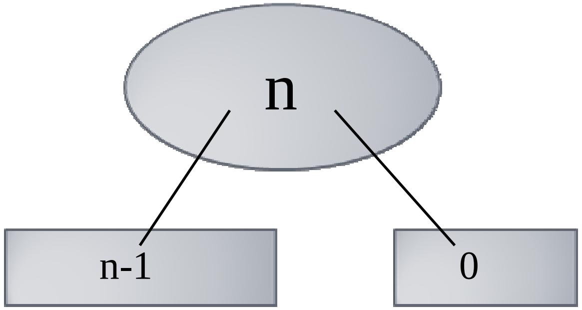
RUNNING TIME OF QSORT [1]

- The time that qsort cost decided by *how we get the partition*.



The best partition (balance)

The worst partition (unbalance)



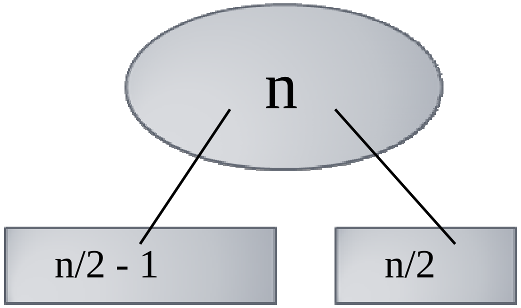
RUNNING TIME OF QSORT [2]

- How can we get if we always in the best partition.

- T(n) = 2T (n/2) + O (n)   (which is approximate to n*logn)
- (note : definition of T(n) and O(n))
- Thus the equal balance of two side at every level of the recursion make faster algorithm.

- Note : We say the time qsort cost is O(n) if

$$\lim \frac{time}{n} = const$$



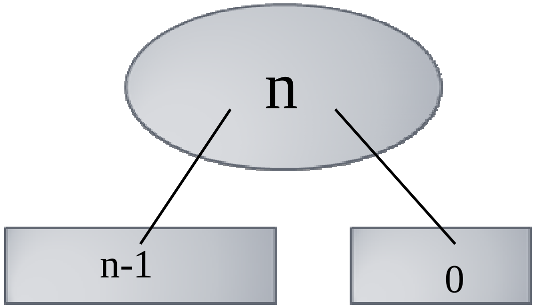
RUNNING TIME OF QSORT [3]

Partition (A,p,r)

```
x ← A[r]
i ← p-1
for  j←p to r-1
     do if  A[j]<=x
          then i←i+1
               exchange A[i]←→A[j]
exchange A[i+1]←→A[r]
return i+1
```

RUNNING TIME OF QSORT [4]

- How can we get if we always in the worse partition.
- T(n) = T (n-1) + O(n) (which is approximate to n^2)
- What will happen if balance and unbalance cases are mixed ?
- Ex : We always get pivot as the largest or the smallest.

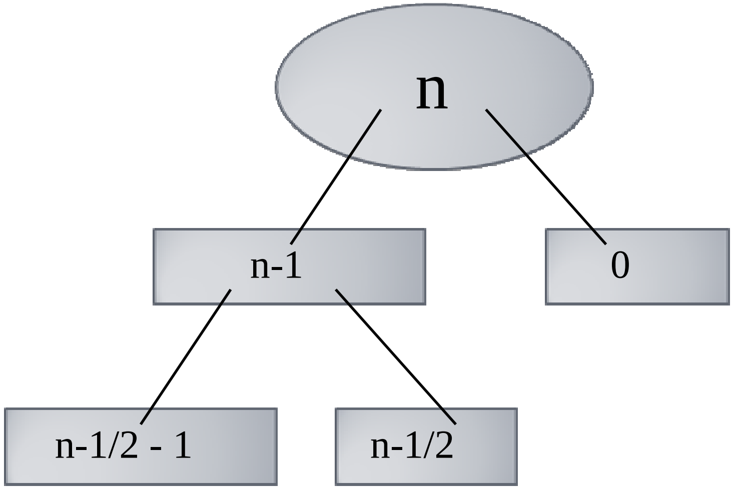


RUNNING TIME OF QSORT [5]

- In the average cast the running time will much closer to good case. Because when bad and good case are consecutive , bad case will be absorbed into good case.

- $T(n) = T(n-1) + T(0) + O(n) \Longrightarrow T(n-1/2 -1) + T(n-1/2) + O(n)$

- However we can't expect we will not meet the unbalance case. How to solve this problem?



RUNNING TIME OF QSORT [6]

- We have know that the average case will close to good case, but how could we make it average ?

```
Randomized-Partition

i  ⟵  Random ( p , r )
exchange A[r]  ⟷  A[i]
return Partition ( A , p , r)
```

Profile of qsot[1]

Red: workstation (g++); Green: vc
Testbench: The worst case (sec)

Workstation	Qsort (no random)		Qsort(random)	
N	time	depth	time	depth
10000	1	9999	0	29
20000	1	19999	0	36
40000	6	39999	0	35
80000	22	79999	0	38
160000	89	159999	0	42
320000	---	---	0	52
640000	---	---	0	70
Visual_C	Qsort (no random)		Qsort(random)	
N	time	depth	time	depth
10000	---	---	0	34
20000	---	---	0	34
40000	---	---	0	43
80000	---	---	0	43
160000	---	---	0	46
320000	---	---	0	47
640000	---	---	0	51

Profile of qsot[2]

Red: workstation (icpc); Green: vc

Testbench: The best case (sec)

Workstation	Qsort (no random)		Qsort(random)		Qsort(C lib)
N	time	depth	time	depth	time
1000000	0	20	1	46	0
2500000	1	22	1	55	0
5000000	1	23	1	52	1
10000000	1	24	2	59	2
20000000	1	25	5	63	5
30000000	2	25	4	63	8
50000000	3	26	8	64	14
Visual_C	Qsort (no random)		Qsort(random)		Qsort(C lib)
N	time	depth	time	depth	time
1000000	0	20	0	91	0
2500000	0	22	2	188	1
5000000	1	23	6	341	3
10000000	3	24	18	633	7
20000000	6	25	69	1238	15
30000000	8	25	147	1840	23
50000000	16	26	406	3059	40

Profile of qsot[3]

Red: workstation (icpc); Green: vc

Testbench: The average case (sec)

Workstation	Qsort (no random)		Qsort(random)		Qsort(C lib)
N	time	depth	time	depth	time
1000000	0	78	0	280	1
2500000	2	135	1	255	1
5000000	4	234	1	258	2
10000000	11	406	3	260	5
20000000	36	740	5	269	10
Visual_C	Qsort (no random)		Qsort(random)		Qsort(C lib)
N	time	depth	time	depth	time
1000000	0	78	0	80	1
2500000	1	135	1	138	3
5000000	4	232	1	229	10
10000000	11	410	3	404	31
20000000	37	753	5	744	162

Get the best case

- The idea of creating the best case is let qsort can get the midterm number to be pivot. As the result, we can always get balance partition.

pseudo code

```
Midterm (A, r, p)

// A is an array which is sorted

A[(p+r)/2 +1]  ↔  A[p]
Sort A[(p+r/2)+1……(p-1)]
mid = [ p + (p+r)/2 +1  ] /2
A[mid]  ↔  A[p/2 +1]
Midterm (A, mid, p)
Partition (A, 0 , mid-1)
```

```
Partition (A, r, p)

//A is an array which is sorted

A[p+r /2]  ↔  A[p]
Sort A[(p+r/2)+1……(p-1)]
Midterm(A, mid, p)
Partition(A, r, mid-1)
```